Dokumentáció

Készítette:
Podmanicky Frigyes

A feladat kitűzése:

Olyan számítógépes program készítése, ami 5-pontos véges-differencia sémát használva parciális differenciálegyenletet old meg numerikusan, 2 dimenzióban, párhuzamosan, vagyis egyszerre több processzort felhasználva, úgy, hogy közben automatikus hálófinomitást végez(AMR).

A megvalósítás:

Az alábbi eszközöket használtam fel:

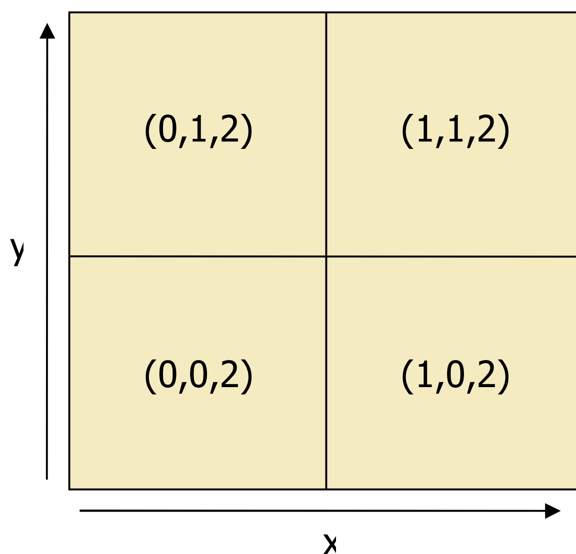
Charm++: http://charm.cs.uiuc.edu/
Visit: https://wci.llnl.gov/codes/visit/home.html
Silo: https://wci.llnl.gov/codes/visit/3rd_party/silo060605.sh
GNU GCC: http://gcc.gnu.org/

A Charm++ környezet a standard C++ programnyelvre épül, e fölött biztosít egy bizonyos absztrakciót. Úgynevezett chare-objektumokra épül, amiket a runtime dinamikusan hoz létre(vagy szüntet meg) egymással hálózatban álló gépeken. Ezekből az objektumokból tömb készíthető. Ez a tömb azonban tetszőleges bit-stringgel indexelhető, és a program futása során lehetőség van elemek beszúrására vagy törlésére. A chare-ek közötti kommunikációt aszinkron függvényhívások teszik lehetővé. Egy chare egy C++ class-nek felel meg, ami egy megfelelő superclass-től öröklődik. Ezt a Charm++ fodító hozza létre egy konfigurációs fájlból(*.ci). Ha egy chare meghívja egy másik chare method-ját, akkor ez a függvényhívás nem blokkolja a hívó függvényt, mert az azonnal visszatér. Viszont létrehoz egy message-t, amit a megfelelő médium továbbít a megfelelő processzornak, annak, amelyiken a hívott chare található. Ennél nagyobb funkcionalitást is biztosít a Charm++ környezet, de ezek voltak szükségesek a program létrehozásához.

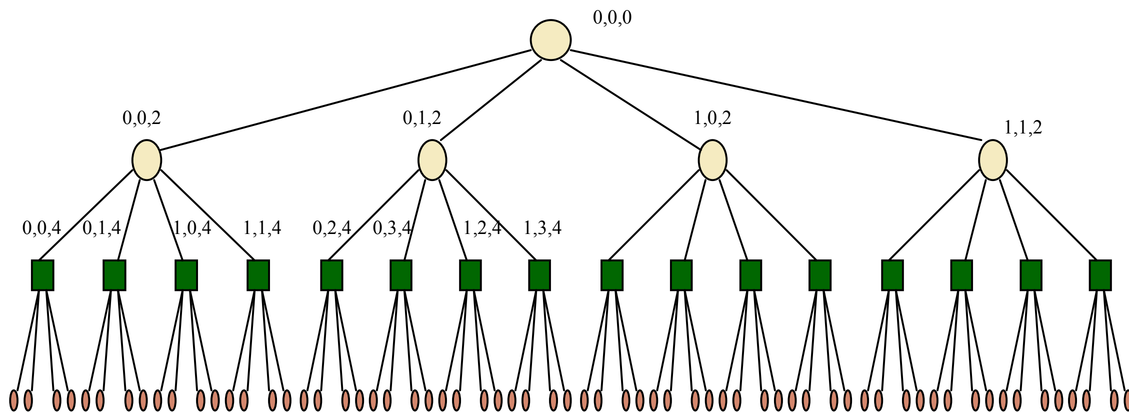
A probléma megoldása a párhuzamosítás feladatával kezdődik. Egy igyen program esetében nyilvánvaló, hogy mit kell tenni: a PDE megoldásának tartományát felosztjuk egyenlő részekre, mivel ezeken egymástól majdnem függetlenül tudjuk alkalmazni a véges-differencia sémát, a 2D-s négyzetek széleit ugyanis szinkronizálnunk kell minden iteráció után. A program tehát úgy működhetne, hogy létrehozunk egy fentebb említett chare-tömböt, aminek minden egyes eleme a tartomány egy négyzetét reprezentálja, a rajta található

adatokkal, és az azon végrehajtandó függvényekkel együtt. Minden időlépésre végrehajtjuk a séma alkalmazását a tartományon(patch), majd kicseréljük a patch-ek legkülső sorait-oszlopait az adott patch szomszédaival. Majd jön a következő időlépés. Mivel mi ezt adaptív hálófinomítással szeretnénk egybekötni, ezért adott időközönként megvizsgáljuk egy feltétel teljesülését minden egyes patch-en. Három kimenet lehetséges: finomítás, durvítás, nincs változás. Ha az adott patch finomítást állapít meg, akkor létrehoz maga helyett 4 másik patch-et, amik ugyanazt a tartományt fedik le, mint amit ő maga, de a lineáris felbontás megkétszerezésével. Durvítás esetén ez fordítva működik, 4 patch alakulhat át 1 ujjá, a felbontás csökkenése mellett. Ez az egész folyamat a következő kényszer figyelembevétele mellett zajlik: szomszédos patch-ek finomítási szintjei között nem lehet a különbség 1-nél nagyobb. Vagyis egy (n)*(n)-es négyzet nem lehet szomszédja egy (4n)*(4n)-nek.

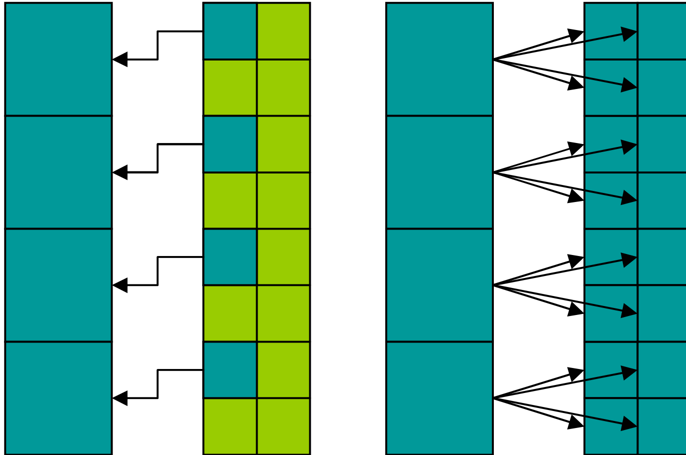
A patch-ek logikai struktúrája egy fa-diagrammot alkot, az indexelési séma a következő(x,y,n): a gyökér indexe: (0,0,0). Az x vagy y irányban haladva 1-gyel növekszik az adott index értéke. A harmadik index megadja, hogy hány db ilyen felbontású patch-csel lehet lefedni a teljes tartományt. A három szám együtt minden patch-hez egyedi indexet rendel. Ha megszeretnénk kapni egy patch-hez tartozó, finomítás utáni új patch-ek indexeit, a következő fomulákat alaklmazhatjuk: n′=n+2, x1′=2*x1+0, y1′=y1+0, x2′=2*x2+1, y2′=y2+0, x3′=2*x3+0, y3′=y3+1, x4′=2*x4+1, y4′=y4+1.



A fa-struktúrát szemlélteti a következő ábra:



Itt a zöld negyzetek jelképezik az aktív patch-eket(node), a sárgák ún. root-patch-eket, a pirosak pedig a leaf-patch-eket. A program futása során ezek mindig jelen vannak. Ez alatt azt értem, hogy az adott tartományhoz tartozó megoldás értékeit a zöld színnel reprezentáltak hordozzák, a másik két típusú patch-re a különböző finomsági fokkal rendelkező patch-ek széleinek szinkronizációjához van szükség. Minden aktív patch-hez tartozik pontosan 4db leaf, és ha nem első generációs patch, akkor egy root. Ha egy patch el szeretné küldeni a legszélső oszlopát/sorát a vele határos patch-nek, akkor egyszerűen a saját indexéből a fenti formulák felhasználásával létrehozza a szomszédja feltételezett indexét. Ehhez az indexhez tartozó patch mindenképpen létezik, ezt a finomitásra vonatkozó kényszer biztosítja. A küldő patch üzenetet küld a címzettnek, amit az feldolgoz, annak megfelelően, hogy a 3 típusból melyikbe esik. Ha root, akkor a továbbküldi az eggyel finomabb szinten lévő node-nak, ha node, akkor egyszerűen feldolgozza, ha leaf, akkor pedig az eggyel durvább szinten node-nak lévőnek küldi tovább. Látható, hogy ha nem node-node kommunikáció játszódik le, akkor a küldő-fogadó oldalon a tartomány felbontása különbözik. Így tehát illeszteni kell az adatokat egymáshoz. A program jelenlegi verziójában ez 0-ad rendű interpolációval van megoldva. Ez természetesen gyakorlati problémák megoldása esetén nem használható, csak demonstrációs célokat szolgál.



A fenti ábra a 0-ad rendű interpolációt demonstrálja, akkor, amikor egy node finomít/durvít. Gyakorlatilag az egymást fedő patch-ek értékeit veszik fel a megfelelő patch-ek. A szélek illesztése ehhez hasonlóan történik, csupán azzal a különbséggel, hogy csak egy oszlopot/sort veszünk figyelembe.

A demonstrációs problémának a térben, időben másodfokú hullámegyenletet választottam, periodikus határfeltételekkel:
$u_{tt}(x,y,t)=u_{xx}(x,y,t)+u_{yy}(x,y,t)$, $u(0,y,t)=u(1,y,t)$, $u(x,0,t)=u(x,1,t)$.
A kezdeti feltételek szabadon beállíthatók u_{ij}^{n}-re és u_{ij}^{n-1}-re. A PDE átírása véges differenciákra az alábbi fomulára vezet:
$u_{ij}^{n+1}=2u_{ij}^{n}-u_{ij}^{n-1}+k\text{^}2/h\text{^}2*(u_{i-1j-1}^{n}+u_{i-1j+1}^{n}+u_{i+1j-1}^{n}+u_{i+1j+1}^{n}+4u_{ij}^{n})$

A program futását egy switch() irányítja. Ha egy chare(a tömb egy eleme) befejezett egy ciklust, akkor meghívja a mainchare egy bizonyos függvényét. Ez a ciklus lehet az adatok mentése, a tartomány frissítése(következő időlépés kiszámítása), finomítás/durvítás. A mainchare tudja, hogy minden egyes iterációban hány olyan patch van, ami aktív(node, vagyis értelmesek a fenti fogalmak, rendelkezik adatmezővel), és addig vár, amíg a tömb összes eleme visszaigazolja a ciklus végét. Ha ez megtörtént, akkor elindítja a következőt. Ez olyan függvényhívással történik, ami a tömb összes elemén végrehajtódik(lehetőség van specializációram csak adott indexű elemen való végrehajtásra is).

A program adatait a Visit nagytudású vizualizációs software-rel dolgoztam fel. Ezt a programot széleskörben alkalmazzák, több tucat tudományos vagy elterjedt adatformátumot képes olvasni. Én a Silo formátumot választottam, mert viszonylag egyszerűen használható, és biztosította azt a funkcionalitást, amire szükségem volt. A mentés menete a következő: minden Charm++ program rendelkezik egy mainchare objektummal, amit a 0. processzoron hoz létre a runtime. Mentéskor a patch-ek ennek az objektumnak elküldik az adataikat, és ez pedig a Silo soros könyvtár függvényhívásaival lemezre menti. Az adatok mellett

szükség van még a négyzetháló pontjainak meghatározására, tehát azokra a koordinátákra, amiken az adott érték található. Ezt is ekkor számítja ki.

A program használata:

A program egyelőre periodikus határfeltételekkel működik. A [0,1]^2 tartományt osztja fel négyzetekre. Ezeknek a száma szabadon beállítható. Megadható a kívánt időlépések száma, a mentések gyakorisága, a finomítás/durvítás gyakorisága, és a finomítási szintek alsó-felső határa. A program fordításához szükség van a Silo könyvtár lefordítására, a Charm++ környezet fordítására(nem feltétlenül szükséges fordítani, néhány operációs rendszerhez van futtatható, letölthető változat), és egy működőképes GNU GCC telepítésre(más fordító is használható). A forráskód mellett van egy Makefile, ezért ha a fenti feltételek teljesülnek, akkor egy egyszerű make paranccsal fordíthatjuk a programot. Ha azt szeretnénk, hogy több processzort használjon, a következő paranccsal futtassuk: ./charmrun ++local +pn ./my_amr. Itt n a processzorok számát jelenti. A paraméterek megváltoztatása a forráskód módosítását, és újrafordítást igényel.

A forrás tehát 4 fájlból áll. ”Makefile” a fordítást egyszerűsíti le, ”my_amr.ci” a Charm++ interfész-fájl, tartalmazza azokat a C++ class-eket, amiket chare-objektumként szeretnénk használni, azokkal a függvényekkel együtt, amiket meg szeretnénk hívni távolról. Fordításkor a Charm++ fordító ennek alapján létrehoz két fájlt, a my_amr.decl.h és my_amr.def.h fájlokat, amiket a forrásunkban includolni kell. ”my_amr.h” tartalmazza a program összes class-definícióját. ”my_amr.C” pedig az ezekhez tartozó kódot.

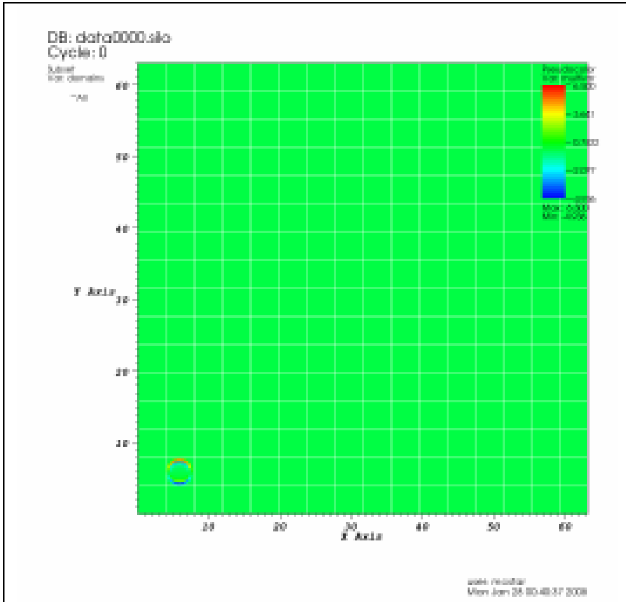
Skálázódás:

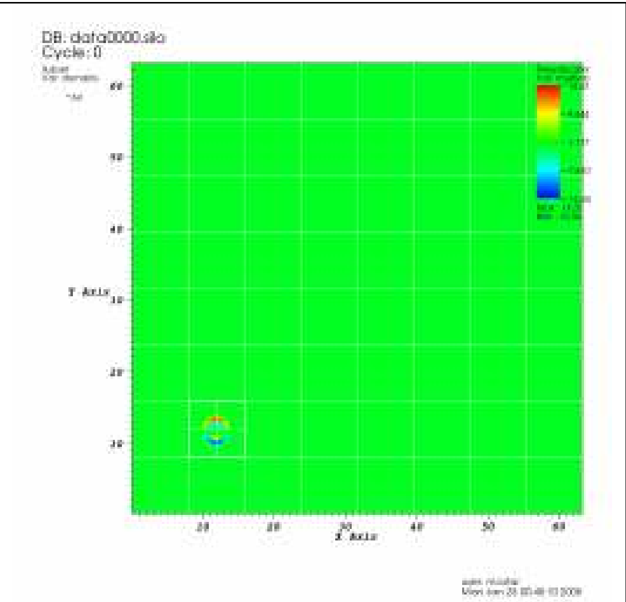
Négy próbafutás eredménye:

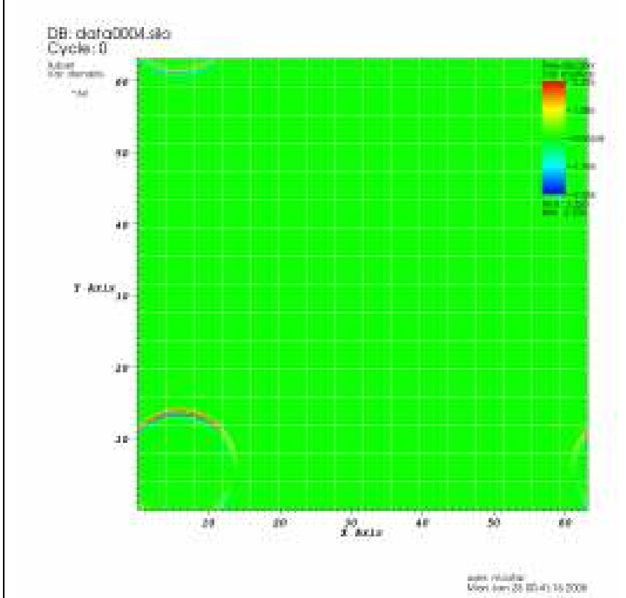
szintek	8	8	6-8-10	6-8-10
#p	1	2	1	2
felbontás	64x64	64x64	64x64	64x64
időlépések	3000	3000	2000	2000
futásidő	411s	266s	544s	348s

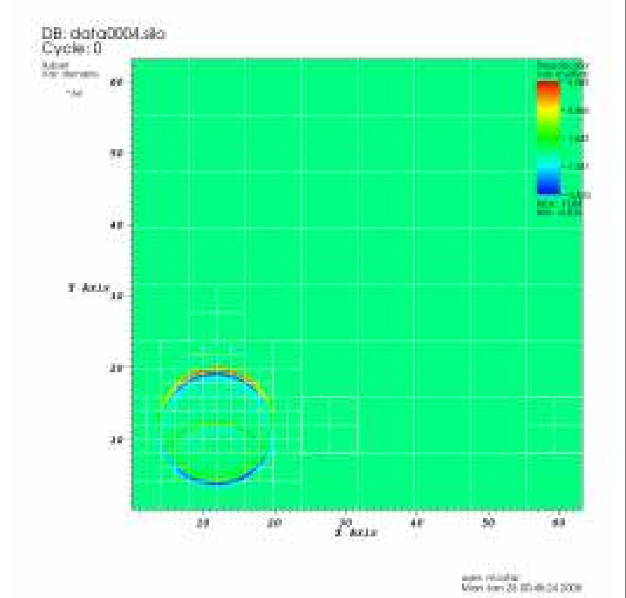
Eredmények:

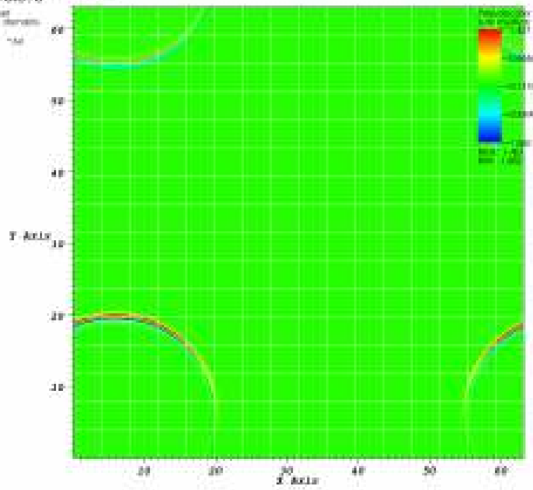
Egy adaptív finomítást használó, és egy konstans felbontású futás összehasonlítása:

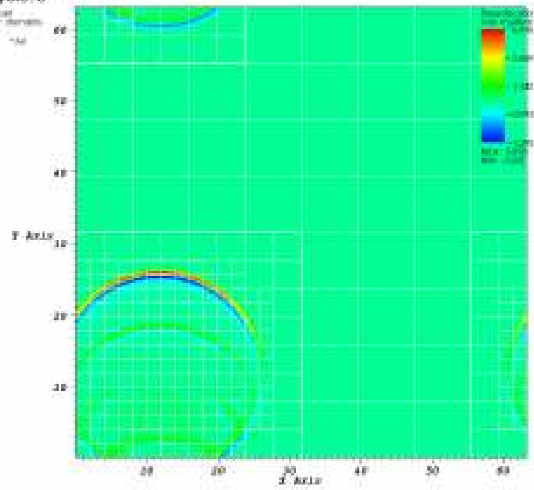

DB: data0000.silo
Cycle: 0
Y Axis
X Axis
10
20
30
40
50
60

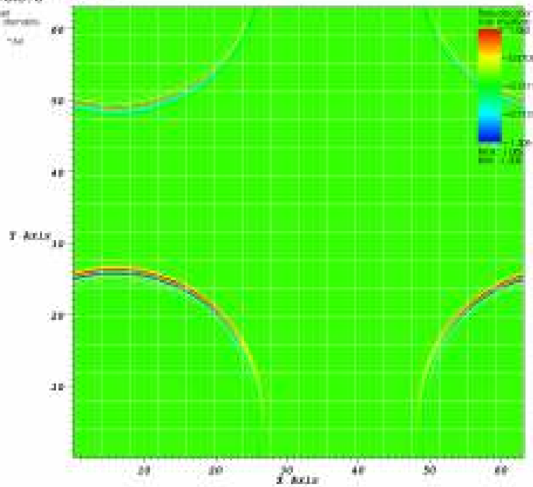

DB: data0000.silo
Cycle: 0
Y Axis
X Axis
10
20
30
40
50
60

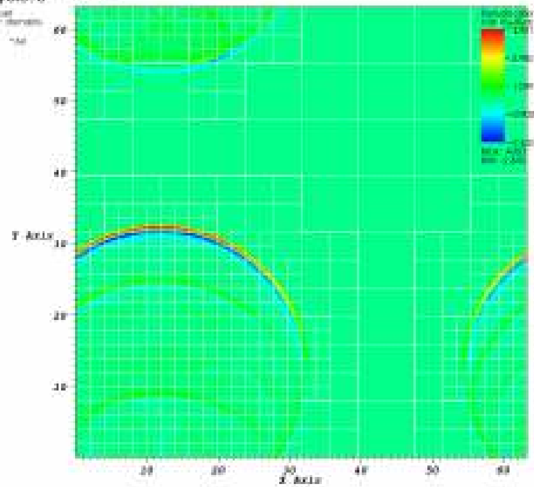

DB: data0004.silo
Cycle: 0
Y Axis
X Axis
10
20
30
40
50
60

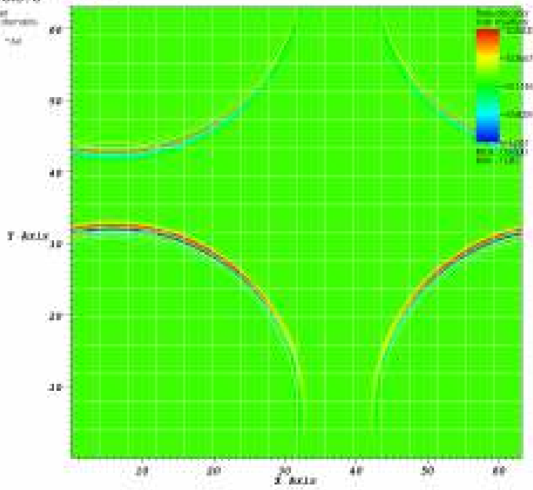

DB: data0004.silo
Cycle: 0
Y Axis
X Axis
10
20
30
40
50
60

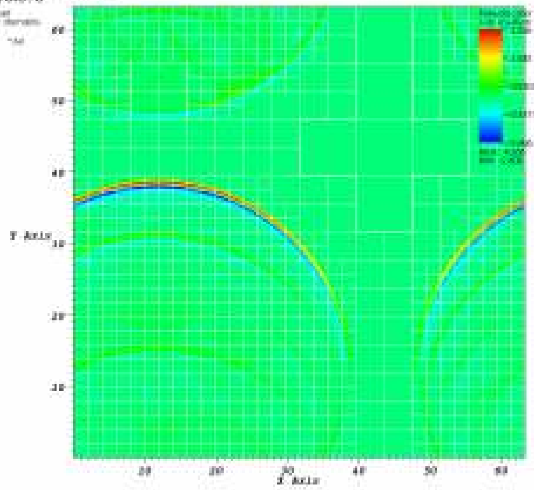

DB: data0008.silo
Cycle: 0
Y Axis
X Axis

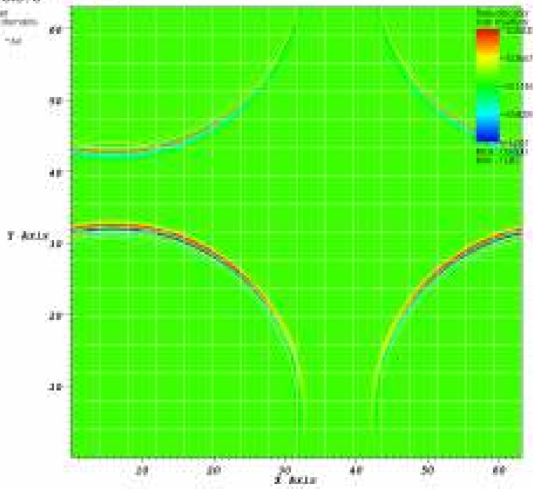

DB: data0008.silo
Cycle: 0
Y Axis
X Axis

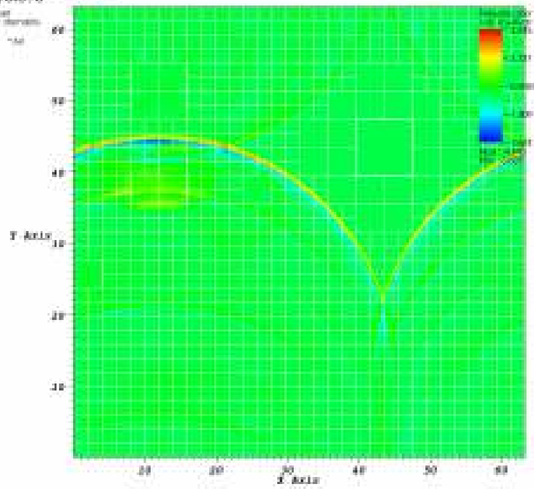

DB: data0012.silo
Cycle: 0
Y Axis
X Axis


DB: data0012.silo
Cycle: 0
Y Axis
X Axis


DB: data0016.silo
Cycle: 0
Y Axis
X Axis


DB: data0016.silo
Cycle: 0
Y Axis
X Axis


DB: data0016.silo
Cycle: 0
Y Axis
X Axis


DB: data0020.silo
Cycle: 0
Y Axis
X Axis

DB: data0020.silo
Cycle: 0

Y Axis

X Axis

DB: data0024.silo
Cycle: 0

Y Axis

X Axis